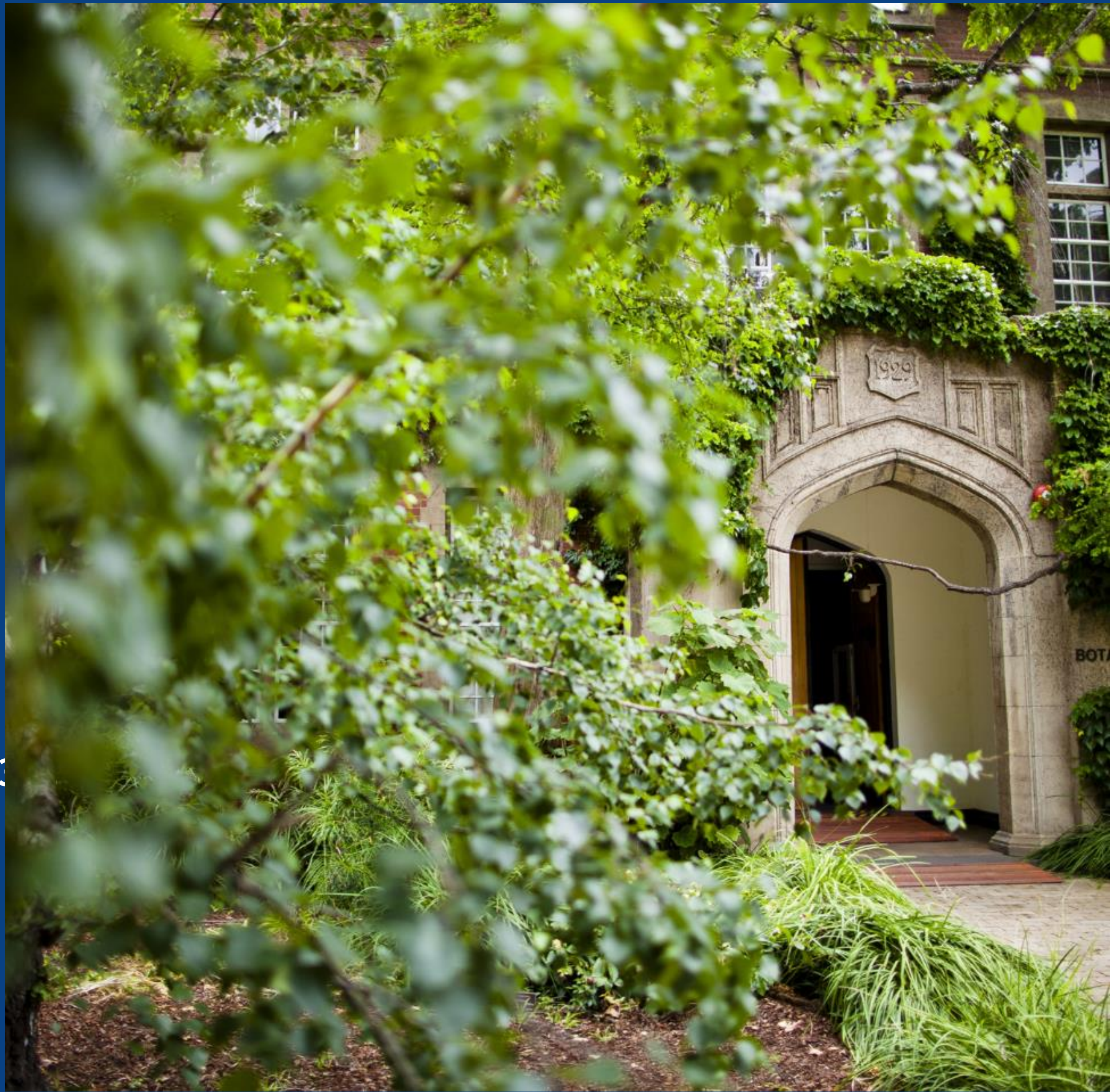
COMP 90048 Declarative Programming Workshop 10 (week11)

2019 semester 1

by Wendy Zeng

Tutorial : Tue 18:15 - 19:15 221 Bouverie St, room B113

Wed 17:15 - 18:15 201 Bouverie St, room B132



Outline

1. Higher Order Function Recap
2. Monad Intro: return and bind operators
3. Monad Example: Maybe
4. Monad Example: IO and Do Block

1. Higher Order Function Recap

- Some useful higher order functions:

- any :: (a -> Bool) -> [a] -> Bool
  - returns True if any of the items in the list fulfill the condition
- all :: (a -> Bool) -> [a] -> Bool
  - returns True if all items in the list fulfill the condition
- flip :: (a->b->c) -> b -> a -> c
  - swap order of the first two parameters

1. Higher Order Function Recap

The function

allSingleton :: [[a]] -> Bool

is intended to check whether all the sublists of its argument are singleton lists. Considering the all, any and flip functions described in lectures, which one of the following is **not** a correct definition of allSingleton?

- allSingleton = all ((==1).length)
- allSingleton = not . (any ((/=1).length))
- allSingleton yss = flip all yss (\xs -> length xs == 1)
- allSingleton = all id (map (==1) . map length)
  - **Fix: allSingleton = all id $ (map (==1) . map length)**
- allSingleton = all $ (==1).length

2. Monad Intro: return and bind operators

- Monad:
  - Monad as a computation: that combines sequence of computation A and B, which depends on the result of A
  - Monad as a container: that contains a type variable a and wraps it within a context
  - Monad is a type class, Monad m => m a indicates m is a type constructor
- return :: Monad m => a -> m a
  - To build a value of this monadic type
- (>>=) :: Monad m => m a -> (a -> m b) -> m b
  - Sequencing / bind operator
  - Chaining sequence of two computations
    - 1. unwrap the value of a from monadic type m a
    - 2. build up a new monadic type m b from the value of a
- (>>) :: Monad m => m a -> m b -> m b
  - Sequencing / bind operator
  - The second computation doesn't require the input from previous computation

3. Monad Example: Maybe

- Maybe:
  - Computation that may not return a value (Nothing)
  - Return operator:
    - return x = Just x
  - Binding operator: for some computation f that takes the input of type a
    - Nothing >>= f = Nothing
    - Just a >>= f = f a

    (f :: a -> Maybe b)

3. Monad Example: Maybe

- Why use the sequencing operator:
  - Don't need to explicitly check for the case for Nothing and Just x
  - Sequential flow or computations that captures the result from previous monadic action, and move on to the next

- Example1: *maybe_drop :: :: Int -> [a] -> Maybe [a]* (drops the first n elements from list, making use of the maybe_tail function)

Without sequencing:

```
maybe_drop1 :: Int -> [a] -> Maybe [a]
maybe_drop1 0 xs = Just xs
maybe_drop1 n list  =
    let mt = maybe_tail list in
    case mt of
        Nothing -> Nothing
        Just tail -> maybe_drop1 (n-1) tail
```

Using >>= :

```
maybe_drop2 :: Int -> [a] -> Maybe [a]
maybe_drop2 0 xs = Just xs
maybe_drop2 n list
    | n > 0 = maybe_tail list >>= maybe_drop1
(n-1)
    | otherwise = Nothing
```

3. Monad Example: Maybe

Consider the term:

(>>= _ -> Just False)

Which one of the following best describes the term's type?

- Monad a -> Maybe Bool
- Monad m => m a -> Maybe Bool
- Maybe Bool -> Maybe Bool
- Maybe a -> Maybe Bool
- *The term is erroneous. It has no valid type because the bind operator requires two arguments.*

(a -> m b) : second param is provided as (_ -> Just False)
Which indicates that b is of type Bool, m is of type class Maybe
Requires first param as m Maybe a, and produces Maybe Bool

4. Monad Example: IO and Do Block

- getChar :: IO Char
- getLine :: IO String

- putChar :: Char -> IO ()
- putStr :: String -> IO ()
- putStrLn :: String -> IO ()
- print :: (Show a) => a -> IO ()

- Do Block:
  - Build a sequence of IO actions
  - What is allowed in do block:

```
do
    x1 <- act1          ---IO action that produces value and bind it to x1
    act2                ---IO action that doesn't produce any value ()
    x2 <- act3
    let x3 = f x1 x2    ---bind non-monadic value x3
    ....
```

4. Monad Example: IO and Do Block

- Why use do block:

```
do
    x1 <- act1
    act2
    x2 <- act3
    let x3 = f x1 x2
....
```

- <- is the generator that we used in list comprehension, creates binding for variables

VS

```
act1 >>=
\x1 ->
        act2 >>=
        \_ ->
                act3
                \x2 -> let x3 = f x1 x2
```

4. Monad Example: IO and Do Block

- Why use do block:

- Example2: *sum_lines :: IO Int* (continuously read a line of string and convert to integer, in the meantime sum up the numbers read)

Without do block:

```
sum_lines1 :: IO Int
sum_lines1 =
    getLine >>=
    \line -> case str_to_num line of
        Nothing -> return 0
        Just num ->
            sum_lines1 >>=
            \sum -> return (sum+num)
```

Using do block :

```
sum_lines2 :: IO Int
sum_lines2 = do
    line <- getLine
    case str_to_num line of
        Nothing -> return 0
        Just num -> do
            sum <- sum_lines2
            return (num+sum)
```

Thank you

wendy.zeng@unimelb.edu.au

By Wendy Zeng